Chapter 2
Branching

Branching is one of the basic algorithmic techniques for designing fast exponential time algorithms. It is safe to say that at least half of the published fast exponential time algorithms are branching algorithms. Furthermore, for many NP-hard problems the fastest known exact algorithm is a branching algorithm. Many of those algorithms have been developed during the last ten years by applying techniques like Measure & Conquer, quasiconvex analysis and related ones.

Compared to some other techniques for the design of exact exponential time algorithms, branching algorithms have some nice properties. Typically branching algorithms need only polynomial (or linear) space. The running time on some particular inputs might be much better than the worst case running time. They allow various natural improvements that do not really change the worst case running time but significantly speed up the running time on many instances.

Branching is a fundamental algorithmic technique: a problem is solved by decomposing it into subproblems, recursively solving the subproblems and by finally combining their solutions into a solution of the original problem.

The idea behind branching is so natural and simple that it was reinvented in many areas under different names. In this book we will refer to the paradigm as *Branch & Reduce* and to such algorithms as *branching algorithms*. There are various other names in the literature for such algorithms, like splitting algorithms, backtracking algorithms, search tree algorithms, pruning search trees, DPLL algorithms etc. A branching algorithm is recursively applied to a problem instance and uses two types of rules.

- A *reduction rule* is used to simplify a problem instance or to halt the algorithm.
- A *branching rule* is used to solve a problem instance by recursively solving smaller instances of the problem.

By listing its branching and reduction rules such an algorithm is in principle easy to describe, and its correctness is often quite easy to prove. The crux is the analysis of the worst-case running time.

The design and analysis of such algorithms will be studied in two chapters. In this chapter we start with several branching algorithms and analyze their running

F.V. Fomin, D. Kratsch, *Exact Exponential Algorithms*, Texts in Theoretical Computer Science. An EATCS Series, DOI 10.1007/978-3-642-16533-7_2,

time by making use of a simple measure of the input, like the number of variables in a Boolean formulae, or the number of vertices in a graph. More involved techniques, which use more complicated measures and allow a better analysis of branching algorithms, in particular Measure & Conquer, will be discussed in Chap. 6.

2.1 Fundamentals

The goal of this section is to present the fundamental ideas and techniques in the design and analysis of branching algorithms; in particular an introduction to the running time analysis of a branching algorithm, including the tools and rules used in such an analysis.

A typical branching algorithm consists of a collection of branching and reduction rules. Such an algorithm may also have (usually trivial) halting rules. Furthermore it has to specify which rule to apply on a particular instance. Typical examples are preference rules or rules on which vertex to branch; e.g. the minimum degree rule of algorithm `mis1` in Chap. 1. To a large part designing a branching algorithm means establishing reduction and branching rules.

In many branching algorithms any instance of a subproblem either contains a corresponding partial solution explicitly or such a partial solution can easily be attached to the instance. Thus a given algorithm computing (only) the optimal size of a solution can easily be modified such that it also provides a solution of optimal size. For example, in a branching algorithm computing the maximum cardinality of an independent set in a graph G, it is easy to attach the set of vertices already chosen to be in the (maximum) independent set to the instance.

The correctness of a well constructed branching algorithm usually follows from the fact that the branching algorithm considers all cases that need to be considered. A typical argument is that at least one optimal solution cannot be overlooked by the algorithm. Formally one has to show that all reduction rules and all branching rules are correct. Often this is not even explicitly stated since it is straightforward. Clearly sophisticated rules may need a correctness proof.

Finally let us consider the running time analysis. Search trees are very useful to illustrate an execution of a branching algorithm and to facilitate our understanding of the time analysis of a branching algorithm. A *search tree* of an execution of a branching algorithm is obtained as follows: assign the root node of the search tree to the input of the problem; recursively assign a child to a node for each smaller instance reached by applying a branching rule to the instance of the node. Note that we do not assign a child to a node when a reduction rule is applied. Hence as long as the algorithm applies reduction rules to an instance the instance is simplified but the instance corresponds to the same node of the search tree.

What is the running time of a particular execution of the algorithm on an input instance? To obtain an easy answer, we assume that the running time of the algorithm corresponding to one node of the search tree is polynomial. This has to be

guaranteed even in the case that many reduction rules are applied consecutively to one instance before the next branching. Furthermore we require that a reduction of an instance does not produce a simplified instance of larger size. For example, a reduction rule applied to a graph typically generates a graph with fewer vertices.

Under this assumption, which is satisfied for all our branching algorithms, the running time of an execution is equal to the number of nodes of the search tree times a polynomial. Thus analyzing the worst-case running time of a branching algorithm means determining the maximum number of nodes in a search tree corresponding to the execution of the algorithm on an input of size n, where n is typically *not* the length of the input but a natural parameter such as the number of vertices of a graph.

How can we determine the worst-case running time of a branching algorithm? A typical branching algorithm has a running time of $\mathcal{O}^*(\alpha^n)$ for some real constant $\alpha \geq 1$. However except for some very particular branching algorithms we are not able determine the smallest possible α. More precisely, so far no general method to determine the *worst-case* running time of a branching algorithm is available, not even up to a polynomial factor. In fact this is a major open problem of the field. Hence analysing the running time means upper bounding the unknown smallest possible value of α. We shall describe in the sequel how this can be done.

The time analysis of branching algorithms is based on upper bounding the number of nodes of the search tree of any input of size n; and since the number of leaves is at least one half of the number of nodes in any search tree, one usually prefers to upper bound the number of leaves. First a measure for the size of an instance of the problem is defined. In this chapter we mainly consider simple and natural measures like the number of vertices for graphs, the number of variables for Boolean formulas, the number of edges for hypergraphs (or set systems), etc.

We shall see later that other choices of the measure of the size of an instance are possible and useful. The overall approach to analyzing the running time of a branching algorithm that we are going to describe will also work for such measures. This will mainly be discussed in Chap. 6.

Let $T(n)$ be the maximum number of leaves in any search tree of an input of size n when executing a certain branching algorithm. The general approach is to analyse each branching rule separately and finally to use the worst-case time over all branching rules as an upper bound on the running time of the algorithm.

Let b be any branching rule of the algorithm to be analysed. Consider an application of b to any instance of size n. Let $r \geq 2$, and $t_i > 0$ for all $i \in \{1,2,\ldots,r\}$. Suppose rule b branches the current instance into $r \geq 2$ instances of size at most $n-t_1$, $n-t_2$, ... , $n-t_r$, for all instances of size $n \geq \max\{t_i : i = 1,2,\ldots r\}$. Then we call $\mathbf{b} = (t_1,t_2,\ldots t_r)$ the *branching vector* of branching rule b. This implies the linear recurrence

$$T(n) \leq T(n-t_1) + T(n-t_2) + \cdots + T(n-t_r). \tag{2.1}$$

There are well-known standard techniques to solve linear recurrences. A fundamental fact is that base solutions of linear recurrences are of the form c^n for some complex number c, and that a solution of a homogeneous linear recurrence is a

linear combination of base solutions. More precisely, a base solution of the linear recurrence (2.1) is of the form $T(n) = c^n$ where c is a complex root of

$$x^n - x^{n-t_1} - x^{n-t_2} - \cdots - x^{n-t_r} = 0. \tag{2.2}$$

We provide some references to the literature on linear recurrences in the Notes.

Clearly worst-case running time analysis is interested in a largest solution of (2.1) which is (up to a polynomial factor) a largest base solution of (2.1). Fortunately there is some very helpful knowledge about the largest solution of a linear recurrence obtained by analysing a branching rule.

Theorem 2.1. *Let b be a branching rule with branching vector $(t_1, t_2, \ldots t_r)$. Then the running time of the branching algorithm using only branching rule b is $\mathcal{O}^*(\alpha^n)$, where α is the unique positive real root of*

$$x^n - x^{n-t_1} - x^{n-t_2} - \cdots - x^{n-t_r} = 0.$$

We call this unique positive real root α the *branching factor* of the branching vector **b**. (They are also called branching numbers.) We denote the branching factor of $(t_1, t_2, \ldots, t_r)$ by $\tau(t_1, t_2, \ldots, t_r)$.

Theorem 2.1 supports the following approach when analysing the running time of a branching algorithm. First compute the branching factor α_i for every branching rule b_i of the branching algorithm to be analysed, as previously described. Now an upper bound of the running time of the branching algorithm is obtained by taking $\alpha = \max_i \alpha_i$. Then the number of leaves of the search tree for an execution of the algorithm on an input of size n is $\mathcal{O}^*(\alpha^n)$, and thus the running time of the branching algorithm is $\mathcal{O}^*(\alpha^n)$.

Suppose a running time expressed in terms of n is what we are interested in. Using n as a simple measure of the size of instances we establish the desired upper bound of the worst-case running time of the algorithm. Using a more complex measure we need to transform the running time in terms of this measure into one in terms of n. This will be discussed in Chap. 6.

Due to the approach described above establishing running times of branching algorithms, they are of the form $\mathcal{O}^*(\alpha^n)$ where $\alpha \geq 1$ is a branching factor and thus a real (typically irrational). Hence the analysis of branching algorithms needs to deal with reals. We adopt the following convention. Reals are represented by five-digit numbers, like 1.3476, such that the real is rounded appropriately: rounded up in running time analysis (and rounded down only in Sect. 6.4). For example, instead of writing $\tau(2,2) = \sqrt{2} = 1.414213562..$, we shall write $\tau(2,2) < 1.4143$, and this implicitly indicates that $1.4142 < \tau(2,2) < 1.4143$. In general, reals are rounded (typically rounded up) and given with a precision of 0.0001.

As a consequence, the running times of branching algorithms in Chap. 2 and Chap. 6 are usually of the form $\mathcal{O}(\beta^n)$, where β is a five-digit number achieved by rounding up a real. See also the preliminaries on the $\mathcal{O}^*$ notation in Chap. 1.

Properties of branching vectors and branching factors are crucial for the design and analysis of branching algorithms. Some of the fundamental ones will be discussed in the remainder of this section.

We start with some easy properties which follow immediately from the definition.

Lemma 2.2. *Let $r \geq 2$. Let $t_i > 0$ for all $i \in \{1,2,\ldots r\}$. Then the following properties are satisfied:*

1. $\tau(t_1,t_2,\ldots,t_r) > 1$.
2. $\tau(t_1,t_2,\ldots,t_r) = \tau(t_{\pi(1)},t_{\pi(2)},\ldots,t_{\pi(r)})$ *for any permutation* π.
3. *If* $t_1 > t_1'$ *then* $\tau(t_1,t_2,\ldots,t_r) < \tau(t_1',t_2,\ldots,t_r)$.

Often the number of potential branching vectors is unbounded, and then repeated use of property 3. of the previous lemma may remove branchings such that a bounded number of branchings remains and the worst-case branching factor remains the same. This is based on the fact that the branching with vector $(t_1',t_2,\ldots,t_r)$ is worse than the branching with vector $(t_1,t_2,\ldots,t_r)$, thus the latter branching can be discarded. We shall also say that $(t_1',t_2,\ldots,t_r)$ *dominates* $(t_1,t_2,\ldots,t_r)$. In general, dominated branching vectors can be safely discarded from a system of branching vectors.

One often has to deal with branching into two subproblems. The corresponding branching vector (t_1,t_2) is called *binary*. Intuitively the factor of a branching vector can be decreased if it is better balanced in the sense of the following lemma.

Lemma 2.3. *Let i,j,k be positive reals.*

1. $\tau(k,k) \leq \tau(i,j)$ *for all branching vectors* (i,j) *satisfying* $i+j = 2k$.
2. $\tau(i,j) > \tau(i+\varepsilon, j-\varepsilon)$ *for all* $0 < i < j$ *and all* $0 < \varepsilon < \frac{j-i}{2}$.

The following example shows that trying to improve the balancedness of a branching rule or branching vector by changes in the algorithm or other means is a powerful strategy.

$$\begin{aligned}\tau(3,3) = \sqrt[3]{2} \quad &< 1.2600\\ \tau(2,4) = \tau(4,2) &< 1.2721\\ \tau(1,5) = \tau(5,1) &< 1.3248\end{aligned}$$

2 A table of precalculated factors of branching vectors can be helpful, in particular in the case of simple analysis or more generally when all branching vectors are integral. Note that the factors have been rounded up, as it is necessary in (upper bounds of) running times. Table 2.1 provides values of $\tau(i,j)$ for all $i,j \in \{1,2,3,4,5,6\}$. They have been computed based on the linear recurrence

$$T(n) \leq T(n-i) + T(n-j) \;\Rightarrow\; x^n = x^{n-i} + x^{n-j}$$

and the roots of the characteristic polynomial (assuming $(j \geq i)$

$$x^j - x^{j-i} - 1.$$

	1	2	3	4	5	6
1	2.0000	1.6181	1.4656	1.3803	1.3248	1.2852
2	1.6181	1.4143	1.3248	1.2721	1.2366	1.2107
3	1.4656	1.3248	1.2560	1.2208	1.1939	1.1740
4	1.3803	1.2721	1.2208	1.1893	1.1674	1.1510
5	1.3248	1.2366	1.1939	1.1674	1.1487	1.1348
6	1.2852	1.2107	1.1740	1.1510	1.1348	1.1225

Table 2.1 A table of branching factors (rounded up)

So far we have studied fundamental properties of branching vectors. Now we shall discuss a way to combine branching vectors. This is an important tool when designing branching algorithms using simple analysis. We call this operation *addition of branching vectors*.

The motivation is easiest to understand in the following setting. Suppose (i, j) is the branching vector with the largest branching factor of our algorithm. In simple time analysis this means that $\tau(i, j)$ is actually the base of the running time of the algorithm. We might improve upon this running time by modifying our algorithm such that whenever branching with branching vector (i, j) is applied then in one or both of the subproblems obtained, the algorithm branches with a better factor. Thus we would like to know the overall branching factor when executing the branchings together. The hope is to replace the tight factor $\tau(i, j)$ by some smaller value.

The corresponding sum of branching vectors is easy to compute by the use of a search tree; one simply computes the decrease (or a lower bound of the decrease) for all subproblems obtained by the consecutive branchings, i.e. the overall decrease for each leaf of the search tree, and this gives the branching vector for the sum of the branchings. Given that the branching vectors are known for all branching rules applied, it is fairly easy to compute the branching vector of the sum of branchings.

Let us consider an example. Suppose whenever our algorithm (i, j)-branches, i.e. applies a branching rule with branching vector (i, j), it immediately (k,l)-branches on the left subproblem. Then the overall branching vector is $(i+k, i+l, j)$, since the addition of the two branchings produces three subproblems with the corresponding decreases of the size of the original instance. How to find the sum of the branchings by use of a search tree is illustrated in Fig. 2.1. In the second example the branching vectors are $(2,2)$ and then $(2,3)$ on the left subproblem and $(1,4)$ on the right subproblem; the sum is $(4,5,3,6)$.

In Sect. 2.3 we use addition of branching vectors in the design and analysis of an algorithm for MAXIMUM INDEPENDENT SET.

2.2 k-Satisfiability

In this section we describe and analyse a classical branching algorithm solving the k-SATISFIABILITY problem. In 1985 in their milestone paper Monien and Speck-

enmeyer presented a branching algorithm and its worst-case running time analysis [159]. Their paper is best known for the fact that their algorithm has time complexity $\mathcal{O}(1.6181^n)$ for the 3-SATISFIABILITY problem.

Let us recall some fundamental notions. Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of *Boolean variables*. A variable or a negated variable is called a *literal*. Let $L = L(X) = \{x_1, \overline{x_1}, x_2, \overline{x_2}, \ldots, x_n, \overline{x_n}\}$ be the set of literals over X. A disjunction $c = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_t)$ of literals $\ell_i \in L(X)$, $i \in \{1, 2, \ldots t\}$, is called a *clause* over X. As usual we demand that a literal appears at most once in a clause, and that a clause does not contain both a variable x_i and its negation $\overline{x}_i$. We represent a clause c by the set $\{\ell_1, \ell_2, \ldots \ell_t\}$ of its literals. A conjunction $F = (c_1 \wedge c_2 \wedge \cdots \wedge c_r)$ of clauses is called a Boolean formula in *conjunctive normal form* (CNF). We represent F by the set $\{c_1, c_2, \ldots, c_m\}$ of its clauses and call it a *CNF formula*. If each clause of a CNF formula consists of at most k literals then it is called a *k-CNF formula*. By $\emptyset$ we denote the empty formula which is a tautology, and thus satisfiable, by definition. A CNF formula F' is a subformula of a CNF formula F if every clause of F' is a subset of some clause of F.

A *truth assignment* t from X to $\{0,1\}$ assigns Boolean values (0=false, 1=true) to the variables of X, and thus also to the literals of $L(X)$. A CNF formula F is *satisfiable* if there is a truth assignment t such that the formula F evaluates to true, i.e. every clause contains at least one literal which is true. For example, the CNF formula

$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3})$$

is satisfied by the truth assignment $x_1 = \text{true}$, $x_2 = \text{false}$, and $x_3 = \text{false}$.

Satisfiability Problem. In the SATISFIABILITY problem (SAT), we are given a Boolean formula in conjunctive normal form (CNF) with n variables and m clauses. The task is to decide whether the formula is satisfiable, i.e. whether there is a truth assignment for which the formula is true.

k-Satisfiability Problem. If the input of the satisfiability problem is in CNF in which each clause contains at most k literals, then the problem is called the k-SATISFIABILITY problem (k-SAT).

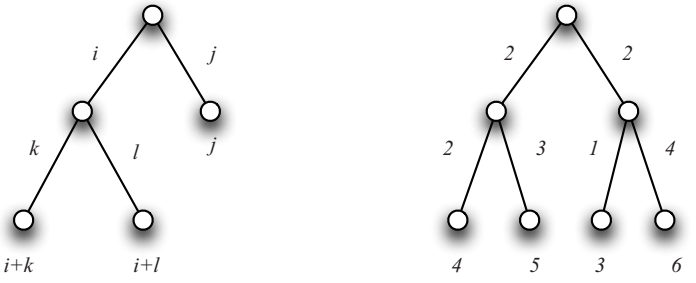

Fig. 2.1 Adding branching vectors

It is well-known that 2-SAT is linear time solvable, while for every $k \geq 3$, k-SAT is NP-complete. How can one decide whether a CNF formula is satisfiable? A trivial algorithm solves SAT by checking every truth assignment of the given CNF formula. If the number of variables is n, then there are 2^n different truth assignments. For each such assignment, we have to check whether every clause is satisfied, which can be done in time $\mathcal{O}(nm)$. Thus the brute-force algorithm will solve the problem in time $\mathcal{O}(2^n nm)$. It is a major open question whether SAT can be solved in time $\mathcal{O}^*((2-\varepsilon)^n)$ for some constant $\varepsilon > 0$.

Now we decribe the branching algorithm of Monien and Speckenmeyer that solves the k-SAT problem in time $\mathcal{O}^*({\alpha_k}^n)$ where $\alpha_k < 2$ for every fixed $k \geq 3$. Let F be a CNF formula with n variables and m clauses such that each clause of F contains at most k literals. Assuming that F does not have multiple clauses, we have that $m \leq \sum_{i=0}^{k} \binom{2n}{i}$, since F has at most $2n$ literals. Consequently the number of clauses of F is bounded by a polynomial in n. Let $X = \{x_1, x_2, \ldots, x_n\}$ be the set of variables of F and $L = \{x_1, \overline{x_1}, x_2, \overline{x_2}, \ldots, x_n, \overline{x_n}\}$ be the corresponding literals. If $L' \subseteq L$ we denote $\mathrm{lit}(L') = \{x_i, \overline{x_i} : x_i \in L' \text{ or } \overline{x_i} \in L'\}$. The size of a clause, denoted $|c|$, is the number of literals of c. By the definition of k-SAT, $|c| \leq k$ for all clauses of F.

The algorithm recursively computes CNF formulas obtained by a partial truth assignment of the input k-CNF formula, i.e. by fixing the Boolean value of some variables and literals, respectively, of F. Given any partial truth assignment t of the k-CNF formula F the corresponding k-CNF formula F' is obtained by removing all clauses containing a true literal, and by removing all false literals. Hence the instance of any subproblem generated by the algorithm is a k-CNF formula and the size of a formula is its number of variables (or the number of variables not fixed by the corresponding partial truth assignment).

We first study the branching rule of the algorithm. Let F be any k-CNF formula and let $c = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_q)$ be any clause of F. Branching on clause c means to branch into the following q subformulas obtained by fixing the Boolean values of some literals as described below:

- F_1 : $\ell_1 = \text{true}$
- F_2 : $\ell_1 = \text{false}, \ell_2 = \text{true}$
- F_3 : $\ell_1 = \text{false}, \ell_2 = \text{false}, \ell_3 = \text{true}$
- $\cdots$ $\cdots$
- F_q : $\ell_1 = \text{false}, \ell_2 = \text{false}, \cdots, \ell_{q-1} = \text{false}, \ell_q = \text{true}$

The branching rule says that F is satisfiable if and only if at least one F_i, $i \in \{1,2,\ldots,q\}$ is satisfiable, and this is obviously correct. Hence recursively solving all subproblem instances F_i, we can decide whether F is satisfiable. The corresponding branching algorithm `k-sat1` is described in Fig. 2.2.

Since the Boolean values of i variables of F are fixed to obtain the instance F_i, $i \in \{1,2,\ldots,q\}$, the number of (non fixed) variables of F_i is $n-i$. Therefore the branching vector of this rule is $(1,2,\ldots,q)$. To obtain the branching factor of $(1,2,\ldots,q)$, as discussed in Sect. 2.1, we solve the linear recurrence

$$T(n) \leq T(n-1) + T(n-2) + \cdots + T(n-q)$$

by computing the unique positive real root of

$$x^q = x^{q-1} - x^{q-2} - \cdots - x - 1,$$

which is equivalent to computing the largest real root of

$$x^{q+1} - 2x^q + 1 = 0.$$

For any clause of size q, we denote the branching factor by β_q. Then, when rounding up, one obtains $\beta_2 < 1.6181$, $\beta_3 < 1.8393$, $\beta_4 < 1.9276$ and $\beta_5 < 1.9660$.

We note that on a clause of size 1, there is only one subproblem and thus this is indeed a reduction rule. By adding some simple halting rules saying that a formula containing an empty clause is unsatisfiable and that the empty formula is satisfiable, we would obtain the first branching algorithm consisting essentially of the above branching rule.

Algorithm `k-sat1`(F).
Input: A CNF formula F.
Output: Return true if F is satisfiable, otherwise return false.

if *F contains an empty clause* **then**
  return false
if *F is an empty formula* **then**
  return true
choose any clause $c = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_q)$ of F
$b_1 =$ `k-sat1`$(F[\ell_1 = \text{true}])$
$b_2 =$ `k-sat1`$(F[\ell_1 = \text{false}, \ell_2 = \text{true}])$
$\cdots$ $\cdots$
$b_q =$ `k-sat1`$(F[\ell_1 = \text{false}, \ell_2 = \text{false}, \cdots, \ell_{q-1} = \text{false}, \ell_q = \text{true}])$
return $b_1 \vee b_2 \cdots \vee b_q$

Fig. 2.2 Algorithm `k-sat1` for *k*-`Satisfiability`

Of course, we may also add the reduction rule saying that if the formula is in 2-CNF then a polynomial time algorithm will be used to decide whether it is satisfiable. The running time of such a simple branching algorithm is $\mathcal{O}^*({\beta_k}^n)$ because for a given k-CNF as an input all instances generated by the branching algorithm are also k-CNF formulas, and thus every clause the algorithm branches on has size at most k.

Notice that the branching factor β_t depends on the size of the clause c chosen to branch on. Hence it is natural to aim at branching on clauses of minimum size. Thus for every CNF formula being an instance of a subproblem the algorithm chooses a clause of minimum size to branch on.

Suppose we can guarantee that for an input k-CNF the algorithm always branches on a clause of size at most $k-1$ (except possibly the very first branching). Such

a branching algorithm would solve k-SAT in time $\mathcal{O}^*({\beta_{k-1}}^n)$. For example, this algorithm would solve 3-SAT in time $\mathcal{O}(1.6181^n)$.

The tool used to achieve this goal is a logical one. Autarkies are partial (truth) assignments satisfying some subset $F' \subseteq F$ (called an autark subset), while not interacting with the clauses in $F \setminus F'$. In other words, a partial truth assignment t of a CNF formula F is called an *autark* if for every clause c of F for which the Boolean value of at least one literal is set by t, there is a literal ℓ_i of c such that $t(\ell_i) = \text{true}$. For example, for the following formula

$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (\overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_4})$$

the partial assignment $x_1 = \text{true}$, $x_2 = \text{false}$ is an autark.

Hence for an autark assignment t of a CNF formula F all clauses for which at least one literal is set by t are simultaneously satisfied by t. Thus if F' is the CNF formula obtained from F by setting Boolean variables w.r.t. t, then F is satisfiable if and only if F' is satisfiable, because no literal of F' has been set by t. On the other hand, if t is not an autark then there is at least one clause c of F which contains a false literal and only false literals w.r.t. t. Hence the corresponding CNF formula F' contains a clause c' with $|c'| < |c|$. Therefore, when a partial truth assignment of a k-CNF formula F is an autark, the algorithm does not branch and is recursively applied to the k-CNF formula F'. When t is not an autark then there is a clause of size at most $k-1$ to branch on.

In the branching algorithm `k-sat2` described in Fig. 2.3 this idea is used as follows. First of all there are two reduction rules for termination: If F contains an empty clause then F is unsatisfiable. If F is an empty formula then F is satisfiable. Then the basic branching rule is adapted as follows. Branch on a clause c of F of minimum size. For each subproblem F_i with corresponding truth assignment t_i verify whether t_i is an autark assignment. If none of the assignments t_i, $i \in \{1,2,\ldots,q\}$ is an autark, then we branch in each of the subproblems as before. However if there is an autark assignment then we recursively solve only the subproblem F' corresponding to this assignment. Indeed, F is satisfiable if and only if F' is satisfiable. This is now a reduction rule.

To complete the running time analysis we notice that whenever branching on a k-CNF formula F one branches on a clause of size at most $k-1$, except possibly for the very first one. Hence the corresponding branching vector is $(1,2,\ldots,q)$ with $q \leq k-1$. Hence the worst case branching factor when executing the algorithm on an input k-CNF formula is $\alpha_k = \beta_{k-1}$. We conclude with the following theorem.

Theorem 2.4. *There is a branching algorithm solving k-SAT with n variables in time $\mathcal{O}^*({\alpha_k}^n)$, where α_k is the largest real root of $x^k - 2x^{k-1} + 1 = 0$. In particular, the branching algorithm solves 3-SAT in time $\mathcal{O}(1.6181^n)$.*

The best known algorithms solving 3-SAT use different techniques and are based on local search. Chapter 8 is dedicated to local search and the SATISFIABILITY problem.

Algorithm `k-sat2`**(F).**
Input: A CNF formula F.
Output: Return true if F is satisfiable, otherwise return false.

```
if F contains an empty clause then
    return false
if F is an empty formula then
    return true
choose a clause c = (ℓ_1 ∨ ℓ_2 ∨ ··· ∨ ℓ_q) of F of minimum size
let t_1 be the assignment corresponding to F_1 = F[ℓ_1 = true]
let t_2 be the assignment corresponding to F_2 = F[ℓ_1 = false, ℓ_2 = true]
···                    ···
let t_q be the assignment corresponding to
F_q = F[ℓ_1 = false, ℓ_2 = false, ··· , ℓ_{q-1} = false, ℓ_q = true]
if there is an i ∈ {1,2,...,q} s.t. t_i is autark for F_i then
    return k-sat2(F_i)
else
    b_1 = k-sat2(F[ℓ_1 = true])
    b_2 = k-sat2(F[ℓ_1 = false, ℓ_2 = true])
    ···                    ···
    b_q = k-sat2(F[ℓ_1 = false, ℓ_2 = false, ··· , ℓ_{q-1} = false, ℓ_q = true])
    if at least one b_i is true then
        return true
    else
        return false
```

Fig. 2.3 Algorithm `k-sat2` for k-SATISFIABILITY

2.3 Independent Set

In this section we present a branching algorithm to compute a maximum independent set of an undirected graph. As in the previous section the running time analysis will be simple in the sense that the size of any subproblem instance is measured by the number of vertices in the graph.

Let us recall that an *independent set* $I \subseteq V$ of a graph $G = (V,E)$ is a subset of vertices such that every pair of vertices of I is non-adjacent in G. We denote by $\alpha(G)$ the maximum size of an independent set of G. The MAXIMUM INDEPENDENT SET problem (`MIS`), to find an independent set of maximum size, is a well-known NP-hard graph problem. A simple branching algorithm of running time $\mathcal{O}^*(3^{n/3}) = \mathcal{O}(1.4423^n)$ to solve the problem `MIS` exactly has been presented and analysed in Chap. 1.

Our aim is to present a branching algorithm to compute a maximum independent set that uses various of the fundamental ideas of maximum independent set branching algorithms, is not too complicated to describe and has a reasonably good running time when using simple analysis.

Our algorithm works as follows. Let G be the input graph of the current (sub)problem. The algorithm applies reduction rules whenever possible, thus branching rules

are applied only if no reduction rule is applicable to the instance. The reduction rules our algorithm uses are the simplicial and the domination rule explained below.

If the minimum degree of G is at most 3 then the algorithm chooses any vertex v of minimum degree. Otherwise the algorithm chooses a vertex v of maximum degree. Then depending on the degree of v, reduction or branching rules are applied and the corresponding subproblem(s) are solved recursively. When the algorithm puts a vertex v into the solution set, we say that it *selects* v. When the algorithm decides that a vertex is not in the solution set, we say that it *discards* v. Before explaining the algorithm in detail, we describe the main rules the algorithm will apply and prove their correctness.

The first one is a reduction rule called the *domination rule*.

Lemma 2.5. *Let $G = (V,E)$ be a graph, let v and w be adjacent vertices of G such that $N[v] \subseteq N[w]$. Then*

$$\alpha(G) = \alpha(G \setminus w).$$

Proof. We have to prove that G has a maximum independent set not containing w. Let I be a maximum independent set of G such that $w \in I$. Since $w \in I$ no neighbor of v except w belongs to I. Hence $(I \setminus \{w\}) \cup \{v\}$ is an independent set of G, and thus a maximum independent set of $G \setminus w$. □

Now let us study the branching rules of our algorithm. The *standard branching* of a maximum independent set algorithm chooses a vertex v and then either it selects v for the solution and solves `MIS` recursively on $G \setminus N[v]$, or it discards v from the solution and solves `MIS` recursively on $G \setminus v$. Hence

$$\alpha(G) = \max(1 + \alpha(G \setminus N[v]), \alpha(G \setminus v)).$$

As already discussed in Chap. 1, this standard branching rule of `MIS` is correct since for any vertex v, G has a maximum independent set containing v or a maximum independent set not containing v. Furthermore if v is selected for any independent set none of its neighbors can be in this independent set. Obviously the branching vector of standard branching is $(d(v)+1, 1)$.

The following observation is simple and powerful.

Lemma 2.6. *Let $G = (V,E)$ be a graph and let v be a vertex of G. If no maximum independent set of G contains v then every maximum independent set of G contains at least two vertices of $N(v)$.*

Proof. We assume that every maximum independent set of G is also a maximum independent set of $G \setminus v$. Suppose there is a maximum independent set I of $G \setminus v$ containing at most one vertex of $N(v)$. If I contains no vertex of $N[v]$ then $I \cup \{v\}$ is independent and thus I is not a maximum independent set, which is a contradiction. Otherwise, let $I \cap N(v) = \{w\}$. Then $(I \setminus \{w\}) \cup \{v\}$ is an independent set of G, and thus there is a maximum independent set of G containing v, a contradiction. Consequently, every maximum independent set of G contains at least two vertices of $N(v)$. □

Using Lemma 2.6, standard branching has been refined recently. Let $N^2(v)$ be the set of vertices at distance 2 from v in G, i.e. the set of the neighbors of the neighbors of v, except v itself. A vertex $w \in N^2(v)$ is called a *mirror* of v if $N(v) \setminus N(w)$ is a clique. Calling $M(v)$ the set of mirrors of v in G, the standard branching rule can be refined via mirrors.

Lemma 2.7. *Let $G = (V,E)$ be a graph and v a vertex of G. Then*

$$\alpha(G) = \max(1 + \alpha(G \setminus N[v]), \alpha(G \setminus (M(v) \cup \{v\}))).$$

Proof. If G has a maximum independent set containing v then $\alpha(G) = 1 + \alpha(G \setminus N[v])$ and the lemma is true. Otherwise suppose that no maximum independent set of G contains v. Then by Lemma 2.6, every maximum independent set of G contains at least two vertices of $N(v)$. Let w be any mirror of v. This implies that $N(v) \setminus N(w)$ is a clique, and thus at least one vertex of every maximum independent set of G belongs to $N(w)$. Consequently, no maximum independent set of G contains a $w \in M(v)$, and thus w can be safely discarded. □

We call the corresponding rule the *mirror branching*. Its branching vector is $(d(v) + 1, |M(v)| + 1)$.

Exercise 2.8. Show that the following claim is mistaken: If G has an independent set of size k not containing v then there is an independent set of size k not containing v and all its mirrors.

Lemma 2.6 can also be used to establish another reduction rule that we call the *simplicial rule*.

Lemma 2.9. *Let $G = (V,E)$ be a graph and v be a vertex of G such that $N[v]$ is a clique. Then*

$$\alpha(G) = 1 + \alpha(G \setminus N[v]).$$

Proof. If G has a maximum independent set containing v then the lemma is true. Otherwise, by Lemma 2.6 a maximum independent set must contain two vertices of the clique $N(v)$, which is impossible. □

Sometimes our algorithm uses yet another branching rule. Let $S \subseteq V$ be a (small) separator of the graph G, i.e. $G \setminus S$ is disconnected. Then for any maximum independent set I of G, $I \cap S$ is an independent set of G. Thus we may branch into all possible independent sets of S.

Lemma 2.10. *Let G be a graph, let S be a separator of G and let $\mathcal{I}(S)$ be the set of all subsets of S being an independent set of G. Then*

$$\alpha(G) = \max_{A \in \mathcal{I}(S)} |A| + \alpha(G \setminus (S \cup N[A])).$$

Our algorithm uses the corresponding *separator branching* only under the following circumstances: the separator S is the set $N^2(v)$ and this set is of size at most 2.

Thus the branching is done in at most four subproblems. In each of the recursively solved subproblems the input graph is $G \setminus N^2[v]$ or an induced subgraph of it, where $N^2[v] = N[v] \cup N^2(v)$.

Finally let us mention another useful branching rule to be applied to disconnected graphs.

Lemma 2.11. *Let $G = (V,E)$ be a disconnected graph and $C \subseteq V$ a connected component of G. Then*

$$\alpha(G) = \alpha(G[C]) \; + \; \alpha(G \setminus C)).$$

A branching algorithm for the MAXIMUM INDEPENDENT SET problem based on the above rules is given in Fig. 2.4.

To analyse algorithm `mis2` let us first assume that the input graph G of the (sub)problem has minimum degree at most 3 and that v is a vertex of minimum degree. If $d(v) \in \{0,1\}$ then the algorithm does not branch. The reductions are obtained by the simplicial rule.
$\mathbf{d(v) = 0}$: then v is selected, i.e. added to the maximum independent set I to be computed, and we recursively call the algorithm for $G \setminus v$.
$\mathbf{d(v) = 1}$: then v is selected, and we recursively call the algorithm for $G \setminus N[v]$.

Now let us assume that $d(v) = 2$.
$\mathbf{d(v) = 2}$: Let u_1 and u_2 be the neighbors of v in G.

(i) $\{u_1,u_2\} \in E$. Then $N[v]$ is a clique and by the simplicial rule $\alpha(G) = 1 + \alpha(G \setminus N[v])$ and the algorithm is recursively called for $G \setminus N[v]$.

(ii) $\{u_1,u_2\} \notin E$. If $|N^2(v)| = 1$ then the algorithm applies a separator branching on $S = N^2(v) = \{w\}$. The two subproblems are obtained either by selecting v and w and recursively calling the algorithm for $G \setminus (N^2[v] \cup N[w])$, or by selecting u_1 and u_2 and recursively calling the algorithm for $G \setminus N^2[v]$. The branching vector is $(|N^2[v] \cup N[w]|, |N^2[v]|)$, and this is at least $(5,4)$. If $|N^2(v)| \geq 2$ then the algorithm applies a mirror branching to v. If the algorithm discards v then both u_1 and u_2 are selected by Lemma 2.6. If the algorithm selects v it needs to solve $G \setminus N[v]$ recursively. Hence the branching vector is $(N^2[v], N[v])$ which is at least $(5,3)$. Thus the worst case for $d(v) = 2$ is obtained by the branching vector $(5,3)$ and $\tau(5,3) < 1.1939$.

Now let us consider the case $d(v) = 3$.
$\mathbf{d(v) = 3}$: Let u_1, u_2 and u_3 be the neighbors of v in G.

(i) $N(v)$ is an independent set: We assume that no reduction rule can be applied. Hence each u_i, $i \in \{1,2,3\}$ has a neighbor in $N^2(v)$, otherwise the domination rule could be applied to v and u_i. Note that every $w \in N^2(v)$ with more than one neighbor in $N(v)$ is a mirror of v.

If v has a mirror then the algorithm applies mirror branching to v with branching vector $(|N[v]|, \{v\} \cup M(v))$ which is at least $(4,2)$ and $\tau(4,2) < 1.2721$.

If there is no mirror of v then every vertex of $N^2(v)$ has precisely one neighbor in $N(v)$. Furthermore by the choice of v, every vertex of G has degree at least 3, and thus, for all $i \in \{1,2,3\}$, the vertex u_i has at least two neighbors in $N^2(v)$. The

Algorithm `mis2`(G).
Input: A graph $G = (V, E)$.
Output: The maximum cardinality of an independent set of G.

if $|V| = 0$ **then**
  return 0
if $\exists v \in V$ *with* $d(v) \leq 1$ **then**
  return $1 + \texttt{mis2}(G \setminus N[v])$
if $\exists v \in V$ *with* $d(v) = 2$ **then**
  (let u_1 and u_2 be the neighbors of v)
  if $\{u_1, u_2\} \in E$ **then**
    return $1 + \texttt{mis2}(G \setminus N[v])$
  if $\{u_1, u_2\} \notin E$ **then**
    if $|N^2(v)| = 1$ **then**
      (let $N^2(v) = \{w\}$)
      return $\max(2 + \texttt{mis2}(G \setminus (N^2[v] \cup N[w])), 2 + \texttt{mis2}(G \setminus N^2[v]))$
    if $|N^2(v)| > 1$ **then**
      return $\max(\texttt{mis2}(G \setminus N[v]), \texttt{mis2}(G \setminus (M(v) \cup \{v\}))$
if $\exists v \in V$ *with* $d(v) = 3$ **then**
  (let u_1u_2 and u_3 be the neighbors of v)
  if $G[N(v)]$ *has no edge* **then**
    if v *has a mirror* **then**
      return $\max(1 + \texttt{mis2}(G \setminus N[v]), \texttt{mis2}(G \setminus (M(v) \cup \{v\}))$
    if v *has no mirror* **then**
      return $\max(1 + \texttt{mis2}(G \setminus N[v]), 2 + \texttt{mis2}(G \setminus N[\{u_1, u_2\}]), 2 + \texttt{mis2}(G \setminus (N[\{u_1, u_3\}] \cup \{u_2\})), 2 + \texttt{mis2}(G \setminus (N[\{u_2, u_3\}] \cup \{u_1\})))$
  if $G[N(v)]$ *has one or two edges* **then**
    return $\max(1 + \texttt{mis2}(G \setminus N[v]), \texttt{mis2}(G \setminus (M(v) \cup \{v\}))$
  if $G[N(v)]$ *has three edges* **then**
    return $1 + \texttt{mis2}(G \setminus N[v])$
if $\Delta(G) \geq 6$ **then**
  choose a vertex v of maximum degree in G
  return $\max(1 + \texttt{mis2}(G \setminus N[v]), \texttt{mis2}(G \setminus v))$
if G *is disconnected* **then**
  (let $C \subseteq V$ be a component of G)
  return $\texttt{mis2}(G[C]) + \texttt{mis2}(G \setminus C)$
if G *is* 4 *or* 5*-regular* **then**
  choose any vertex v of G
  return $\max(1 + \texttt{mis2}(G \setminus N[v]), \texttt{mis2}(G \setminus (M(v) \cup \{v\}))$
if $\Delta(G) = 5$ *and* $\delta(G) = 4$ **then**
  choose adjacent vertices v and w with $d(v) = 5$ and $d(w) = 4$ in G
  return
  $\max(1 + \texttt{mis2}(G \setminus N[v]), 1 + \texttt{mis2}(G \setminus (\{v\} \cup M(v) \cup N[w])), \texttt{mis2}(G \setminus (M(v) \cup \{v, w\})))$

Fig. 2.4 Algorithm `mis2` for MAXIMUM INDEPENDENT SET

algorithm branches into four subproblems. It either selects v or when discarding v it inspects all possible cases of choosing at least two vertices of $N(v)$ for the solution:

- select v
- discard v, select u_1, select u_2
- discard v, select u_1, discard u_2, select u_3
- discard v, discard u_1, select u_2, select u_3

The branching vector is at least $(4,7,8,8)$ and $\tau(4,7,8,8) < 1.2406$.

(ii) The graph induced by $N(v)$ contains one edge, say $\{u_1,u_2\} \in E$: By the choice of v, the vertex u_3 has degree at least 3, and thus at least two neighbors in $N^2(v)$. Those neighbors of u_3 are mirrors. The algorithm applies mirror branching to v and the branching factor is at least $(4,3)$ and $\tau(4,3) < 1.2208$.

(iii) The graph induced by $N(v)$ contains two edges, say $\{u_1,u_2\} \in E$ and $\{u_2,u_3\} \in E$. Thus when mirror branching on v either v is selected and $G \setminus N[v]$ solved recursively, or v is discarded and u_1 and u_3 are selected. Hence the branching factor is at least $(4,5)$ and $\tau(4,5) < 1.1674$.

(iv) If $N(v)$ is a clique then apply reduction by simplicial rule.

Summarizing the case $d(v) = 3$, the worst case for $d(v) = 3$ is the branching vector $(4,2)$ with $\tau(4,2) < 1.2721$. It dominates the branching vectors $(4,3)$ and $(4,5)$ which can thus be discarded. There is also the branching vector $(4,7,8,8)$.

Now assume that the input graph G has minimum degree at least 4. Then the algorithm does not choose a minimum degree vertex (as in all cases above). It chooses a vertex v of maximum degree to branch on it.

$\mathbf{d(v) \geq 6}$: The algorithm applies mirror branching to v. Thus the branching vector is $(d(v)+1,1)$ which is at least $(7,1)$ and $\tau(7,1) < 1.2554$.

$\mathbf{d(v) = 4}$: Due to the branching rule applied to disconnected graphs the graph G is connected. Furthermore G is 4-regular since its minimum degree is 4.

For any $r \geq 3$, there is at most one r-regular graph assigned to a node of the search tree from the root to a leaf, since every instance generated by the algorithm is an induced subgraph of the input graph of the original problem. Hence any branching rule applied to r-regular graphs, for some fixed r, can only increase the number of leaves by a multiplicative constant. Hence we may neglect the branching rules needed for r-regular graphs in the time analysis.

$\mathbf{d(v) = 5}$: A similar argument applies to 5-regular graphs, and thus we do not have to consider 5-regular graphs. Therefore we may assume that the graph G has vertices of degree 5 and vertices of degree 4, and no others since vertices of smaller or higher degrees would have been chosen to branch on in earlier parts of the algorithm. As G is connected there is a vertex of degree 5 adjacent to a vertex of degree 4. The algorithm chooses those vertices to be v and w such that $d(v) = 5$ and $d(w) = 4$. Now it applies a mirror branching on v. If there is a mirror of v then the branching vector is at least $(2,6)$ and $\tau(2,6) < 1.2366$. If there is no mirror of v then mirror branching on v has branching vector $(1,6)$ and $\tau(1,6) < 1.2852$.

To achieve a better overall running time of the branching algorithm than $\Theta^*(1.2852^n)$ we use addition of branching vectors. Note that the subproblem obtained by discarding v in the mirror branching, i.e. the instance $G \setminus v$, contains the

vertex w such that the degree of w in $G \setminus v$ is equal to 3. Now whenever the algorithm does the (expensive) $(1,6)$-branching it immediately branches on w in the subproblem obtained by discarding v. Hence we may replace $(1,6)$ by the branching vectors obtained by addition. The candidates for addition are the branching vectors for branching on a vertex of degree 3. These are $(4,2)$ and $(4,7,8,8)$. Adding $(4,2)$ to $(1,6)$ (in the subproblem achieved by a gain of 1) gives $(5,3,6)$. Adding $(4,7,8,8)$ to $(1,6)$ gives $(5,8,9,9,6)$. The corresponding branching factors are $\tau(5,6,8,9,9) < 1.2548$ and $\tau(3,5,6) < 1.2786$. See Fig. 2.5.

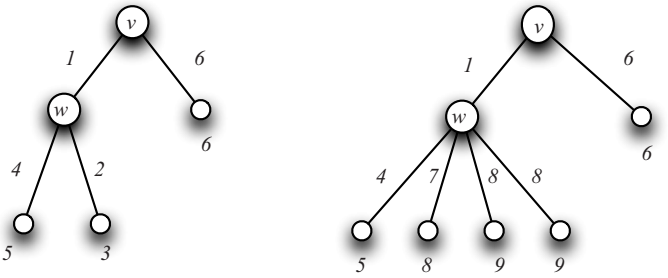

Fig. 2.5 Adding branching vectors in the case $d(v) = 5$

Consequently the worst case branching factor of the whole branching algorithm is $\tau(3,5,6) < 1.2786$.

Theorem 2.12. *The branching algorithm* `mis2` *for the* MAXIMUM INDEPENDENT SET *problem has running time* $\mathcal{O}(1.2786^n)$.

Exercise 2.13. Improve upon the branching algorithm `mis2` by a more careful analysis of the case $d(v) = 3$.

Exercise 2.14. Improve upon the branching algorithm `mis1` of Chap. 1 by the use of Lemma 2.6.

Notes

The analysis of the worst-case running time of branching algorithms is based on solving linear recurrences. This is a subject of most textbooks in Discrete Mathematics. We recommend Rosen [191] and Graham, Knuth, Patashnik [104].

The history of branching algorithms can be traced back to the work of Davis and Putnam [63] (see also [62]) on the design and analysis of algorithms to solve the SATISFIABILITY problem (SAT). It is well-known that SAT and 3-SAT are

NP-complete [100], while there is a linear time algorithm solving 2-SAT [8]. For a few decades branching algorithms have played an important role in the design of exact algorithms for SAT and in particular 3-SAT. One of the milestone branching algorithms for 3-SAT and k-SAT, $k \geq 3$, has been established by Monien and Speckenmeyer in 1985 [159]. In his long paper from 1999 [147], Kullmann establishes a branching algorithm of running time $\mathcal{O}(1.5045^n)$. He also presents a comprehensive study of the worst-case analysis of branching algorithms. See also [148] for an overview. Parts of the Fundamentals section are inspired by [147]. General references for algorithms and complexity of SAT are [21, 130]. Branching algorithms are used to solve different variants of SAT like Max-2-SAT, XSAT , counting variants, and SAT parameterized by the number of clauses [57, 44, 58, 105, 112, 140, 144, 145, 146].

Another problem with a long history in branching algorithms is the MAXIMUM INDEPENDENT SET problem dating back to 1977 and the algorithm of Tarjan and Trojanowski [213]. We refer the reader to the Notes of Chap. 1 for more information.

Many of the reduction and branching rules for MAXIMUM INDEPENDENT SET can be found in [213]. Fürer used the separator rule in [97]. The mirror rule was been introduced by Fomin, Grandoni and Kratsch as part of their Measure & Conquer-based branching algorithm for MAXIMUM INDEPENDENT SET [85], which also uses the folding rule. Kneis, Langer, and Rossmanith in [133] introduced the satellite rule.

Chvátal studied the behaviour of branching algorithms for MIS and showed that for a large class of branching algorithms there is a constant $c > 1$ such that for almost all graphs the running time of these algorithms is more than c^n [50]. Lower bounds for different variants of DPLL algorithms for SAT are discussed in [1, 3, 166].

http://www.springer.com/978-3-642-16532-0

Exact Exponential Algorithms
Fomin, F.V.; Kratsch, D.
2010, XIV, 206 p. 38 illus., Hardcover
ISBN: 978-3-642-16532-0